Using MMX™ Instructions to implement 2X 8-bit Image Scaling

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Using MMX™ Instructions to implement 2X 8-bit Image Scaling

March 1996

CONTENTS

1.0. INTRODUCTION

The MMX^{TM} technology uses the Single Instruction, Multiple Data (SIMD) technique to speed up software, by processing multiple data elements in parallel. This application note exploits the SIMD instructions to implement the 2X image scaling algorithm. The MMX instructions are used to read pixels from the video memory area. After duplicating the pixels in the registers MM0 through MM7, the results are written out to the destination memory area. The end result is a 2X expanded view of the upper left rectangle of the original image. Specifically, the MMX instruction MOVQ is used to transfer packed 64-bit data per cycle to/from memory. The PUNPCKHBW/PUNPCKLBW instructions are used to duplicate multiple pixels in parallel.

The MMX technology implementation is compared with a C implementation and a scalar implementation and the results are summarized.

2.0. 2X IMAGE SCALING

The 2X image scaling algorithm takes an 8-bit image as an input. The algorithm can be defined as follows:

$$Pd\ (x_d, y_d) = Pd\ (x_d+1, y_d) = Pd\ (x_d, y_d+1) = Pd\ (x_d+1, y_d+1) = Ps\ (x_s, y_s)$$

Where:
x_s = horizontal position of the source pixel, x_d = horizontal position of the destination pixel, y_s = vertical position of the source pixel, y_d = vertical position of the destination pixel, Ps = source pixel and Pd = destination pixel.

For each pixel in the source image, the destination image contains four pixels with the same value. The resulting image is a 2X expanded view of the upper left rectangle of the source image.

2.1. C IMPLEMENTATION

Two surfaces are created in the video memory area -- the primary surface and the backbuffer surface. After coping the image to both surfaces, the primary surface is displayed. A pointer to the backbuffer surface memory is passed to the function *image2x* (see listing 1). The two surfaces are flipped once the image is expanded using *image2x* function.

The rest of this section explains the C implementation of the *image2x* function. As the code shows, the *srcptr* pointer is set to the end of the upper left rectangle of the original image described by dimensions (0, 0, x_res/2, y_res/2). The *destptr* pointer points to the end of the surface memory. During each iteration of the inner loop, a pixel is read using strptr pointer and is expanded by writing to four destination locations.

```
/*************************************************************************
*                                                                        *
*       This program has been developed by Intel Corporation. You have      *
*       Intel's permission to incorporate this code into your product,  *
*       royalty free.  Intel has various intellectual property rights   *
*       which it may assert under certain circumstances.                *
*                                                                        *
*       Intel specifically disclaims all warranties, express or          *
*       implied, and all liability, including consequential and other   *
*       indirect damages, for the use of this code, including liability *
*       for infringement of any proprietary rights, and including the   *
*       warranties of merchantability and fitness for a particular      *
*       purpose.  Intel does not assume any responsibility for any      *
*       errors which may appear in this code nor any responsibility to  *
*       update it.                                                           *
*                                                                        *
*    * Other brands and names are the property of their respective       *
*      owners.                                                           *
*                                                                        *
*  Copyright (c) 1996, Intel Corporation.  All rights reserved.          *
*                                                                        *
*************************************************************************/
void image2x (byte *lpBackbufferMemory, int x_res, int y_res )
// lpBackbufferMemory: Points to the beginning of the surface
// x_res: Surface width
```

```
// y_res: Surface height
// NOTE: This program assumes surface width to be equal to the surface
// pitch
{
  byte *srcptr, *destptr;
  int i, j;
  srcptr = ((y_res/2) * x_res) - (x_res/2) + lpBackbufferMemory;
  // srcptr points to the end of the source rectangle to be expanded
  destptr = (y_res * x_res) + lpBackbufferMemory;
  // despptr points to the end of the surface memory
// In the loop below, pixels are processed starting from the high memory
// locations and moving to lower memory locations.
  for (j=y_res/2; j>0; j--)
  {
    for ( i=0; i< (x_res/2); i++  )
    {
      destptr = destptr - 2;
      srcptr = srcptr - 1;
      *destptr = *srcptr;
      *(destptr + 1) = *srcptr;
      *(destptr-x_res) = *srcptr;
      *(destptr+1-x_res) = *srcptr;
    }

    srcptr = srcptr - (x_res/2);
    destptr = destptr - x_res;
  }
}
```

Code Listing 1: C Implementation of the 2X Image Scaling Algorithm

2.2. SCALAR IMPLEMENTATION

Code listing 2 shows the scalar implementation of the function *image2x*.

Notice the main loop is unrolled four times compared to the C version. In each loop iteration, four pixels are read and operated on, as opposed to one in the C implementation.

In this implementation, the pixels are duplicated using two scalar registers before being written to the video memory. A scalar 32-bit register is used to hold two pixels in the lower word, the upper word is cleared. The BSWAP instruction is used to transfer the same pixel data to the upper word of another register. The lower word is cleared. The duplication is complete after both registers are added and the result is rotated 8-bits. The entire duplication process consumes five instructions.

```
TITLE   image2x
;/*************************************************************************
;*       This program has been developed by Intel Corporation. You have  *
;*       Intel's permission to incorporate this code into your product,  *
;*       royalty free.  Intel has various intellectual property rights   *
;*       which it may assert under certain circumstances.                *
;*                                                                       *
;*       Intel specifically disclaims all warranties, express or         *
;*       implied, and all liability, including consequential and other   *
;*       indirect damages, for the use of this code, including liability *
;*       for infringement of any proprietary rights, and including the   *
```

```
;*       warranties of merchantability and fitness for a particular      *
;*       purpose.  Intel does not assume any responsibility for any      *
;*       errors which may appear in this code nor any responsibility to  *
;*       update it.                                                       *
;*                                                                        *
;*    * Other brands and names are the property of their respective       *
;*      owners.                                                           *
;*                                                                        *
;*  Copyright (c) 1996, Intel Corporation.  All rights reserved.          *
;*                                                                        *
;**************************************************************************/
;
;
;*************************************************************************
; prevent listing of iammx.inc file
.nolist
;INCLUDE iammx.inc                   ; IAMMX Emulator Macros
.list
.586
.model FLAT
;*************************************************************************
;    Data Segment Declarations
;*************************************************************************
.data
x_half   DWORD    0H
x2_res   DWORD    0H
cmpptr   DWORD    0H
;*************************************************************************
;    Constant Segment Declarations
;*************************************************************************
.const
;*************************************************************************
;    Code Segment Declarations
;*************************************************************************
.code
;COMMENT ^
;void image2x (
;       BYTE *lpBackbufferMemory,
;       int x_res,
;       int y_res);
;^
; lpBackbufferMemory: Points to the beginning of the surface
; x_res: Surface width
; y_res: Surface height
; NOTE: This program assumes surface width to be equal to the surface
; pitch
image2x PROC NEAR C USES  eax ebx ecx edx edi esi,
            lpBackbufferMemory: PTR BYTE,
            x_res: DWORD,
            y_res: DWORD
        mov         edi, x_res      ; edi = x_res

        mov         edx, x_res
        mov         eax, edi
        mov         esi, edi
        shr         edx, 1          ; edx = x_res /2
```

```
    imul       eax, y_res     ; edx:eax = x_res * y_res

    ; But since x_res * y_res does not exceed 65536 ( individual)
    ; The results in edx are ignored

    mov        ebx, eax
    add        eax, lpBackbufferMemory

    shr        ebx, 1          ; ebx = (y_res/2) * x_res
    sub        eax, edi        ; eax = end destination - x_res

    mov        x_half, edx
    add        ebx, lpBackbufferMemory ; ebx = source end + x_half
    mov        ecx, eax
    sub        ebx, x_half     ; ebx = source end
    shl        esi, 1          ; esi = 2*x_res
    sub        ecx, edi        ; ecx = end destination - 2*x_res
    mov        x2_res, esi


    mov        cmpptr, ecx
    mov        ecx, edi        ; ecx = x_res
    ; Pointer initialization is completed here.
    ; The main loop starts here
loopstart:

    mov        edx, [ebx - 4] ; Read first 32-bit -- from source cache line
    sub        ebx, 4         ; Update pointer ebx and eax

    sub       eax, 8

    mov        esi, edx       ; Copy original data [4321] to esi and edi
    mov        edi, edx       ; where [4321] each no. represent byte position
                              ; using little-endian convention
    bswap      esi            ; esi = [1234]
    and        edx, 0000ffffh ; edx = [0021]
    and        esi, 0ffff0000h; esi = [1200]
    add        esi, edx       ; esi = 1221
    mov        edx, edi       ; edx = [4321]

    bswap      edx            ; edx = [1234]
    rol        esi, 8         ; esi = 2211 -- result no. 1
    mov        [eax], esi     ; write result no. 1 to the first row

    and        edi, 0ffff0000h; edi = [4300]
    and        edx, 0000ffffh ; edx = [0034]

    mov        [eax + ecx], esi ; write result no. 1 to the second row
    add        edi, edx       ; edi = [4334]
    ror        edi, 8         ; edi = [4433] -- result no. 2
    cmp        cmpptr, eax
    mov        [eax + 4], edi       ; write result no. 2 to the first row
    mov        [eax + ecx + 4], edi ; write result no. 2 to the second row

    jne        loopstart
    mov        edx, x2_res
```

```
        sub         cmpptr, edx
        sub         eax, x_res

        sub         ebx, x_half
        cmp         eax, lpBackbufferMemory

        ja          loopstart
    ret
image2x ENDP
END
```

Code Listing 2: Scalar Implementation of the 2X Image Scaling Algorithm

2.3. MMX TECHNOLOGY IMPLEMENTATION

The MMX technology implementation of the function *image2x* exploits the use of 64-bit registers and the MMX instructions to operate in parallel on multiple pixels. Using the MOVQ instruction, eight pixels are loaded in a single cycle to a 64-bit register. Once data is copied to another register, the PUNPCKLBW/PUNPCKHBW instructions are used to unpack the pixels. The results are stored to the video memory using the MOVQ instruction (see figure 1). Since the above described operations use only two registers, a block of 8 x 4 pixels can be operated on in a single iteration using all 64-bit registers.

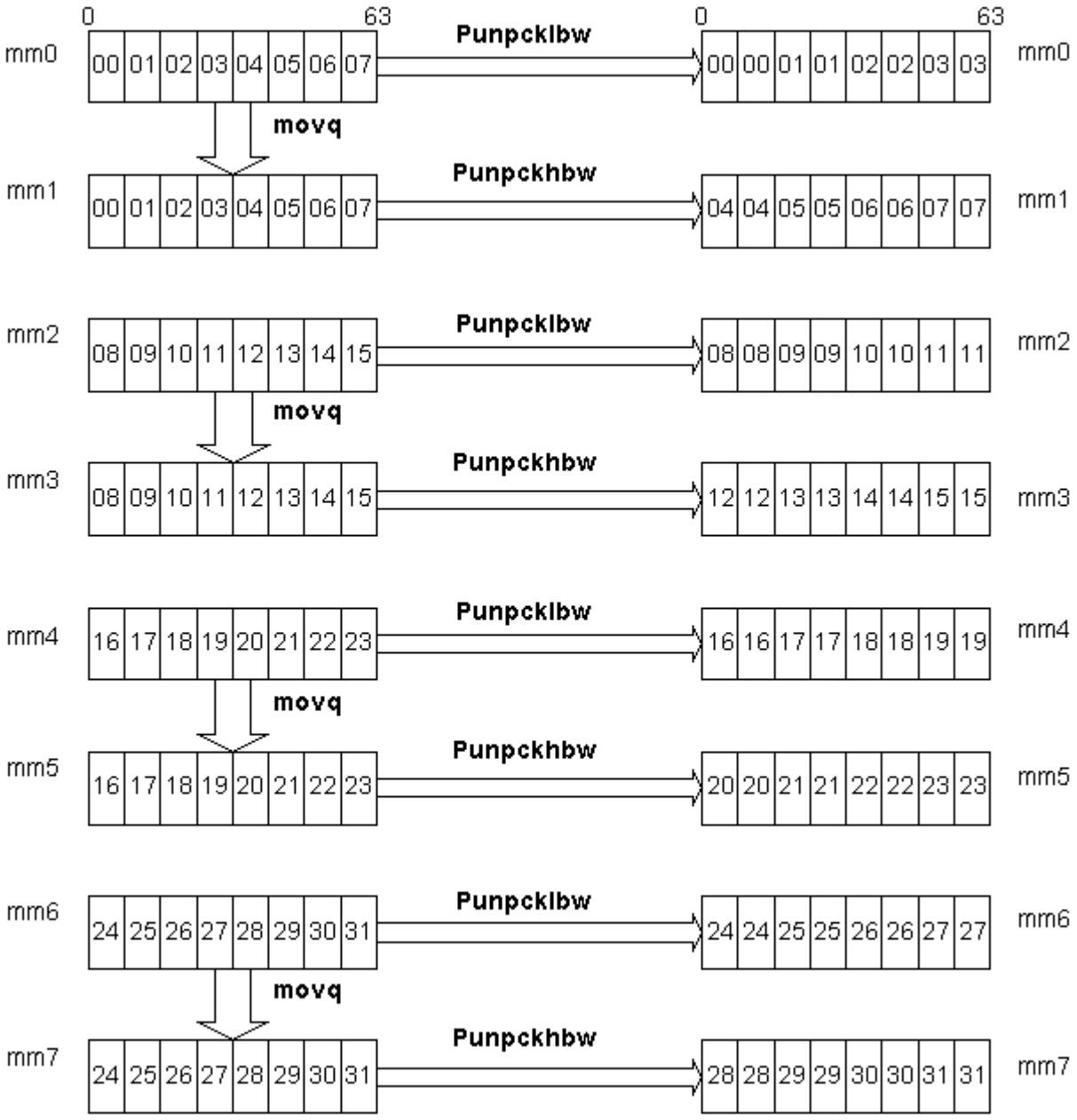

Figure 1: MMX Technology implementation of the 2X Image Scaling Algorithm

Code listing 3 shows the function utilizing MMX instructions. Major implementation differences are highlighted below:

- The availability of eight 64-bit registers permits operation on a 8 x 4 pixel block in a single loop iteration. This results in eight times more unrolling of the loop when compared to the scalar implementation. The book-keeping cycles are reduced and better pairing of instructions is achieved due to unrolling.
The availability of registers also reduces the use of memory cycles because less variables are defined and stored in the memory.

- The PUNPCKHBW/PUNPCKLBW instructions are used to duplicate pixels. As shown in the previous section, similar operation implemented in the scalar code consumes five instructions.

- The use of the MOVQ instruction results in a transfer of 64-bit of data per cycle, reducing instructions in the main loop when compared with the scalar implementation.

Both the scalar version and the MMX technology implementation have been optimized. Several optimization techniques are listed below:

- To avoid excessive write buffer stalls during memory write cycles, selected destination memory locations are first read to bring in a cache line and then data written out to the memory location. This results in better performance since cache line writes are faster than four 64-bit individual writes.

- All memory access are 64-bit aligned.

- The long latency read cycles from the main memory are scheduled first followed by the write operations. Writes before reads are avoided as this results in poor performance.

```
TITLE   image2x
;/*************************************************************************
;*       This program has been developed by Intel Corporation. You have  *
;*       Intel's permission to incorporate this code into your product,  *
;*       royalty free.  Intel has various intellectual property rights   *
;*       which it may assert under certain circumstances.                *
;*                                                                       *
;*       Intel specifically disclaims all warranties, express or         *
;*       implied, and all liability, including consequential and other   *
;*       indirect damages, for the use of this code, including liability *
;*       for infringement of any proprietary rights, and including the   *
;*       warranties of merchantability and fitness for a particular      *
;*       purpose.  Intel does not assume any responsibility for any      *
;*       errors which may appear in this code nor any responsibility to  *
;*       update it.                                                      *
;*                                                                       *
;*    * Other brands and names are the property of their respective      *
;*      owners.                                                          *
;*                                                                       *
;*  Copyright (c) 1996, Intel Corporation.  All rights reserved.         *
;*                                                                       *
;*************************************************************************/
;
;
;
;*************************************************************************
; prevent listing of iammx.inc file
.nolist
INCLUDE iammx.inc                   ; IAMMX Emulator Macros
.list
.586
.model FLAT
;*************************************************************************
;    Data Segment Declarations
;*************************************************************************
.data
```

```
x_half   DWORD    0H
;*************************************************************************
;    Constant Segment Declarations
;*************************************************************************
.const
;*************************************************************************
;    Code Segment Declarations
;*************************************************************************
.code
;COMMENT ^
;void image2x (
;       BYTE *lpBackbufferMemory,
;       int x_res,
;       int y_res);
;^
; lpBackbufferMemory: Points to the beginning of the surface
; x_res: Surface width
; y_res: Surface height
; NOTE: This program assumes surface width to be equal to the surface
; pitch
image2x PROC NEAR C USES  eax ebx ecx edx edi esi,
            lpBackbufferMemory: PTR BYTE,
            x_res: DWORD,
            y_res: DWORD
         mov        edi, x_res     ; edi = x_res

        mov         edx, x_res
        mov         eax, edi
        shr         edx, 1          ; edx = x_res /2
        mov         esi, edi

        imul        eax, y_res     ; edx:eax = x_res * y_res

        ; But since x_res * y_res does not exceed 65536 ( individual)
        ; The results in edx are ignored

        mov         ebx, eax
        add         eax, lpBackbufferMemory

        shr         ebx, 1          ; ebx = (y_res/2) * x_res
        sub         eax, edi        ; eax = end destination - x_res

        mov         x_half, edx
        add         ebx, lpBackbufferMemory ; ebx = source end + x_half
        mov         ecx, eax
        sub         ebx, x_half     ; ebx = source end
        shl         esi, 1          ; esi = 2*x_res
        sub         ecx, edi        ; ecx = end destination - 2*x_res

        ; Pointer initialization is completed here.
        ; The main loop starts here
    loopstart:

        movq        mm0, [ebx - 32]

        movq        mm1, mm0
        mov         edx, [eax - 64] ; Cache read -- better performance
```

```
sub         ebx, 32
mov         edx, [eax - 32] ; Cache read -- better performance

sub         eax, 64
punpcklbw   mm0, mm0

movq        mm2, [ebx + 8]  ; 2nd read
punpckhbw   mm1, mm1

movq        [eax + edi], mm0    ; 2a write -- row no. 2

movq        [eax + edi + 8], mm1 ; 2b write -- row no. 2
movq        mm3, mm2
movq        [eax], mm0      ; 1a write -- row no. 1
punpcklbw   mm2, mm2
movq        [eax + 8], mm1  ; 1b write

movq        [eax + 16], mm2     ; 1c write

punpckhbw   mm3, mm3
movq        mm4, [ebx + 16] ; 3rd read

movq        [eax + 24], mm3 ; 1d write
movq        mm5, mm4

movq        [eax + edi + 16], mm2 ; 2c write
punpcklbw   mm4, mm4
movq        [eax + edi + 24], mm3 ; 2d write
punpckhbw   mm5, mm5

movq        [eax + 32], mm4 ; 1e write


movq        mm6, [ebx + 24] ; 4th read

movq        [eax + 40], mm5 ; 1f write

movq        mm7, mm6

movq        [eax + edi + 32], mm4 ; 2e write
punpcklbw   mm6, mm6
movq        [eax + edi + 40], mm5 ; 2f write
punpckhbw   mm7, mm7

movq        [eax + 48], mm6 ; 1g write

movq        [eax + 56], mm7 ; 1h write

movq        [eax + edi + 48], mm6 ; 2g write

movq        [eax + edi + 56], mm7 ; 2h write
cmp         ecx, eax
jne         loopstart

sub         ecx, esi
sub         eax, edi
```

```
        sub         ebx, x_half
        cmp         eax, lpBackbufferMemory

        ja          loopstart
    emms
    ret
image2x ENDP
END
```

Code Listing 3: MMX Technology Implementation of the 2X Image Scaling Algorithm

3.0. PERFORMANCE RESULTS

All performance analysis is done using Intel's vTune 2.0 Beta 3.0 visual tuning software. A pointer to a 640 x 480 x 8 resolution image stored in the video memory is given as an input to each of the routines listed below.

Table 1: Performance Results Using Intel's vTune Visual Tuning Software

	C Routine	**Optimized Scalar Routine**	**Optimized MMX Technology Routine**
# of Instructions Executed	1, 076, 902	423, 860	92, 417
Total Cycle Count	2, 005, 828	341, 352	148, 791
Count Per Instruction	1.86	0.81	1.68
% Pairing	99.98%	90.94%	73.77%
Performance Gain compared to C Implementation	1X	5.9X	13.5X
Performance Gain compared to Scalar Implementation	NA	1X	2.3X

NOTES:

1) C routine is compiled using Microsoft Visual C++ with the compiler options set to produce Pentium code and optimization set to maximum speed.
2) MMX technology routine assembled with MASM 6.11d.
3) Performance gain compared to C implementation = (Total Cycle Count of C routine) / Total Cycle Count.
4) Performance Gain compared to Scalar Implementation = (Total Cycle Count of Optimized Scalar Routine)/ Total Cycle Count.
5) Results listed in the table are obtained using dynamic analysis environment of the Intel's vTune Visual Tuning Software.

4.0. CONCLUSION

This application note has shown a successful use of MMX instructions to implement a 2X image scaling algorithm. The MMX technology implementation demonstrated greater than two times performance gain when compared to the scalar implementation. The gain can be attributed to the additional availability of eight 64-bit registers, efficient MMX instructions to manipulate pixels at byte level and the assembly level code optimization.

Although the MMX technology routine presented in this application note strictly applies to 8-bit images. The implementation can easily be changed to suit greater color depth images.